

Preparazione alla prova di esame

Alcuni esercizi svolti

Esempio prova di teoria

- A Le variabili condizione: definizione, uso e possibili semantiche dell'operazione signal.
- B Organizzazione fisica del file system Unix.
- C Si consideri la seguente porzione di codice di un programma di sistema Unix:

```
int pid1=100,pid2=100, k=1;
pid1 = fork();
if (pid1) k--;
pid2 = fork();
if (pid2==0)
{ printf("Juventus %d \n", k);
  k++;
}
if (!k) printf("Barcelona\n");
printf("RISULTATO: %d \n", k);
```

In condizioni di funzionamento ideale (senza errori nelle fork, printf, ...), quanti processi complessivamente vengono creati? Quali sono le relazioni gerarchiche tra i processi creati? quale output produce ogni processo?

System calls

Esercizio 1 system calls

Si scriva un programma in C che, utilizzando le *system call* di unix, preveda la seguente sintassi:

esame N1 N2 f C1 C2

dove:

esame è il nome dell'eseguibile da generare

- **N1, N2** sono interi positivi
- **f** e` il nome di un file
- **C1, C2** sono singoli caratteri

Il comando dovrà funzionare nel modo seguente:

- il processo ‘padre’ **P0** deve creare 2 processi figli: **P1 e P2**;

- **ciascun figlio Pi (i=1,2) dovrà accedere al file f in lettura, per “campionare” dal file 1 carattere ogni Ni e confrontarlo con il carattere Ci dato come argomento. Se il carattere “campionato” risulta uguale a Ci, Pi dovrà notificare in modo asincrono l’evento al padre PO.**
- **una volta creati i 2 figli, il padre PO si sospende in attesa di notifiche da parte dei figli: per ogni notifica rilevata, PO dovrà scrivere il pid del processo che l’ha trasmessa in un file binario di nome “notifiche” e risosprendersi.**

Il primo figlio che termina la lettura del file dovrà provocare la terminazione dell’intera applicazione.

Soluzione dell'esercizio

```
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>
int PID1, PID2, fd;

void gestore_F(int sig);
void gestore_T(int sig);

main(int argc , char *argv[])
{
    int N1, N2;
    char C1, C2;
    char S1[80], S2[80];
```

```
if (argc!=6)
{ printf("sintassi sbagliata!\n");
    exit(1);
}

N1=atoi(argv[1]);
N2=atoi(argv[2]);
C1=argv[4][0];
C2=argv[5][0];
signal(SIGUSR1, gestore_F);
signal(SIGUSR2, gestore_F);
signal(SIGQUIT, gestore_T);
fd=open("notifiche", O_WRONLY);
PID1=fork();
```

```
if (PID1==0) /*codice figlio P1*/
{
    fd=open(argv[3], O_RDONLY);
    while (read(fd, S1, N1)>0)
        if (S1[0]==C1)
            kill(getppid(), SIGUSR1);
    kill(getppid(), SIGQUIT);
    exit(0);
}
else if (PID1<0) exit(-1);
PID2=fork();
if (PID2==0)
{/*codice figlio P2*/
    fd=open(argv[3], O_RDONLY);
    while (read(fd, S2, N2)>0)
        if (S2[0]==C2)
            kill(getppid(), SIGUSR2);
    kill(getppid(), SIGQUIT);
    exit(0);
}
```

```
else if (PID2<0) exit(-1);
while(1) // attesa P0
    pause();
}

void gestore_F(int sig)
{ int PID;
printf("%d: ricevuto %d!\n", getpid(), sig);
if (sig==SIGUSR1)
    PID=PID1;
else
    PID=PID2;
write(fd, &PID, sizeof(int));
return;
}
```

```
void gestore_T(int sig)
{ printf("%d: ricevuto segnale di
terminazione!\n", getpid());
    close(fd);
    kill(0, SIGKILL);
    return;
}
```

Esercizio 2 system call

Testo

Si realizzi un programma, che, utilizzando le *system call* del sistema operativo Unix, soddisfi le seguenti specifiche:

Sintassi di invocazione:

esame filein Comando Cstop Cecc

Significato degli argomenti:

- **esame**: nome dell'eseguibile generato.
- **filein**: nome di un file leggibile.
- **Comando**: nome di un file eseguibile.
- **Cstop, Cecc**: singoli caratteri.

Comportamento:

Il processo iniziale (**P0**) deve creare un processo figlio (**P1**).

- P1 dovrà leggere il contenuto del file **filein**, e trasferirlo integralmente al processo padre P0.

- Il processo PO , una volta creato il processo figlio P1, dovrà leggere e stampare sullo standard output quanto inviatogli dal processo figlio P1, secondo le seguenti modalità:
 - Ogni carattere letto diverso da Cstop e da Cecc, viene stampato da PO sullo standard output;
 - Nel caso in cui PO legga il carattere **Cstop**, dovrà semplicemente terminare forzatamente l'esecuzione di entrambi i processi;
 - Nel caso in cui PO legga il carattere **Cecc**, PO dovrà interrompere l'esecuzione del figlio P1; P1 dal momento dell'interruzione in poi, passerà ad eseguire il comando Comando, e successivamente terminerà.

Soluzione dell'esercizio

```
#include <stdio.h>
#include <signal.h>
#include <fcntl.h>

int pp[2];
char com[20];
void trap(int num);

main(int argc, char **argv)
{ int pid0, pid1, fd, k, status;
  char filein[20], buf[40], c;
```

```
if (argc!=5)
{ printf("Sintassi errata!! - esame
filein Comando Cstop Cecc\n");
exit();
}
strcpy(com,argv[2]);
pipe(pp);
signal(SIGUSR1, trap);
pid1=fork();
```

```
if (pid1==0) /*codice figlio */
{ close(pp[0]);
  fd=open(argv[1], O_RDONLY);
  if (fd<0)
  { perror("open");
    exit();
  }
  while( (k=read(fd, &buf, 40))>0)
  { printf("FIGLIO: ho letto %s\n", buf);
    write(pp[1],&buf, k);
  }
  close(fd);
  close(pp[1]);
  exit();
}
```

```
else if (pid1>0) /* codice padre */
{ signal(SIGUSR1, SIG_DFL);
close(pp[1]);
while((k=read(pp[0], &C, 1))>0)
{ printf("PADRE: %c\n", C);
if(C==argv[3][0]) kill(0, SIGKILL); //Cstop
else if (C==argv[4][0])
{ kill(pid1, SIGUSR1); //Cecc
close(pp[0]);
wait(&status);
exit();
}
else write(1, C, 1); //altro
}
wait(&status);
close(pp[0]);
exit(0);
}
```

```
else
{ perror("creazione!") ;
  exit() ;
}
}/* fine main*/
```

```
void trap(int num)
{
  close(pp[1]) ;
  execlp(com, com, (char *) 0) ;
  exit(-1) ;
}
```

Esercizio 3 System Call

Testo

Si scriva un programma in C che realizzi un comando che, utilizzando le system call di unix, preveda la seguente sintassi:

esame file_in car N1 N2

dove:

- esame è il nome dell'eseguibile da realizzare
- file_in è il nome di un file esistente, su cui si hanno i diritti di lettura
- car è un carattere
- N1 e N2 sono interi positivi

Il comando dovrà funzionare nel modo seguente:

- il processo ‘padre’ (P0) deve creare un processo figlio, P1.
- il processo ‘figlio’ (P1) deve creare un processo nipote, P2;
- il padre P0 deve leggere il contenuto di **file_in**: ogni volta che incontra il carattere **car** all’interno del file, ne deve comunicare al figlio e al nipote la posizione all’interno del file (numero intero positivo);
- il figlio P1, per ogni valore V ricevuto da P0, confronta il valore di V con N1; al raggiungimento o eventuale superamento del valore N1, P1 deve avvisare il padre, provocare la terminazione del padre e di P2, e infine terminare.
- il nipote P2, per ogni valore V ricevuto da P0, confronta il valore di V con N2; al raggiungimento (o eventuale superamento) del valore N2, P1 deve avvisare P0, provocare la terminazione di P0 e di P1 e infine terminare.

Impostazione

- Comunicazione dei processi figlio e nipote con P0: uso di due pipe:

pipe1 // com. tra P0 e P1

Pipe2 // com. tra P0 e P2

- uso dei segnali.

Soluzione :

```
#include <fcntl.h>
#include <signal.h>
void handler1(int sig); /*gestore segnali figlio */
void handler2(int sig); /*gestore segnali nipote */
int ppid, fpid, npid;

main(int argc, char **argv)
{
    int pos, pipe1[2], pipe2[2];
    char n, car;
    int N1, N2, val, fd;
    if (argc!=5)
    {   printf("sintassi!\n");
        exit(-1);
    }
    ppid=getpid();
    N1=atoi(argv[3]); N2=atoi(argv[4]);
    if (pipe(pipe1)<0) exit(-2);
    if (pipe(pipe2)<0) exit(-2);
    signal(SIGUSR1, handler1);
    signal(SIGUSR2, handler2);
```

```
if ((fpid=fork())<0) /*creaz. figlio*/
{ perror("fork"); exit(-3); }
else if (fpid==0) /* figlio*/
{ if((npid=fork())<0) /* creaz. nipote*/
{ perror("fork"); exit(-3); }
else if (npid==0) /* nipote*/
{
    close(pipe1[0]);
    close(pipe1[1]);
    close(pipe2[1]);
    sleep(1);
    while(n=read(pipe2[0], &pos,sizeof(int))>0)
    { printf("NIPOTE: ricevuto %d\n", pos);
      if(pos>=N2)
      {
          kill(ppid,SIGUSR2);
          close(pipe2[0]);
          exit(0);
      }
    }/* fine while*/
    close(pipe2[0]); exit(0);
} /* fine nipote */
```

```
else /*figlio*/
{
    close(pipe2[0]);
    close(pipe2[1]);
    close(pipe1[1]);
    sleep(1);
    while(n=read(pipe1[0], &pos,sizeof(int))>0)
    {
        printf("FIGLIO: ricevuto %d\n", pos);
        if(pos>=N1)
        {
            kill(ppid,SIGUSR1);
            kill(npid,SIGKILL);
            close(pipe1[0]);
            exit(0);
        }
    }
    close(pipe1[0]); exit(0);
} /* fine figlio */
```

```
/* padre: */
close(pipe1[0]);
close(pipe2[0]);
if ((fd=open(argv[1],O_RDONLY))<0)
{perror("apertura file"; exit(-5);}
pos=0;
while((n=read(fd,&car, 1))>0)
{ if(car==argv[2][0])
  {   write(pipe1[1], &pos, sizeof(int));
      write(pipe2[1], &pos, sizeof(int));
  }
  pos++;
}
close(fd); close(pipe1[1]); close(pipe2[1]);
} /* fine main*/
```

```
void handler1(int sig) /*gest.segnali figlio */
{ printf("segnale %d dal figlio %d!\n", sig, fpid);
exit(0);
}
```

```
void handler2(int sig) /*gestore segnali nipote */
{ printf("segnale %d dal nipote %d!\n", sig, npid);
kill(fpid, SIGKILL);
exit(0);
}
```

Esercizio 4: system calls

Si scriva un programma in C che, utilizzando le *system call* di unix, preveda la seguente sintassi:

esame N N1 N2 C

dove:

esame è il nome dell'eseguibile da generare

- N, N1, N2 sono interi positivi
- C e` il nome di un file eseguibile (presente nel PATH)

Il comando dovrà funzionare nel modo seguente:

- il processo ‘padre’ P0 deve creare 2 processi figli: P1 e P2;

Il comando dovrà funzionare nel modo seguente: il processo ‘padre’ P0 deve creare 2 processi figli: P1 e P2;

- **il figlio P1 deve aspettare N1 secondi e successivamente eseguire il comando C;**
- **il figlio P2 dopo N2 secondi dalla sua creazione dovrà provocare la terminazione del processo fratello P1 e successivamente terminare; nel frattempo P2 deve periodicamente sincronizzarsi con il padre P0 (si assuma la frequenza di 1 segnale al secondo).**
- **il padre P0, dopo aver creato i figli, si pone in attesa di segnali da P2: per ogni segnale ricevuto, dovrà stampare il proprio pid; al N-simo segnale ricevuto dovrà attendere la terminazione dei figli e successivamente terminare.**

Soluzione dell'esercizio

```
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>
int PID1, PID2,N,esci=0;
int cont=0; /* cont. dei segnali ricev. da P0*/
void gestore_P(int sig); /* gestore di SIGUSR1
                           per P0*/
void timeout(int sig); /* gestore timeout P2*/
main(int argc , char *argv[])
{
    int N1, N2, pf, status, i;
    char com[20];
```

```
if (argc!=5)
{ printf("sintassi sbagliata!\n");
  exit(1);
}

N=atoi(argv[1]);
N1=atoi(argv[2]);
N2=atoi(argv[3]);
strcpy(com, argv[4]);
signal(SIGUSR1, gestore_P);
PID1=fork();
```

```
if (PID1==0) /*codice figlio P1*/
{
    sleep(N1);
    execlp(com,com,(char *)0);
    exit(0);
}
else if (PID1<0) exit(-1);

PID2=fork();
if (PID2==0)
{ /*codice figlio P2*/
    int pp=getppid();
    signal(SIGALRM, timeout);
    alarm(N2);
    for(;;)
    { sleep(1); kill(pp, SIGUSR1); }
    exit(0);
}
```

```
else if (PID2<0) exit(-1);
/* padre */
while(1) pause();

exit(0);

}
```

```
void gestore_P(int sig)
{    int i, status, pf;
    cont++;
    printf("padre %d: ricevuto %d (cont=%d) !\n", getpid(),
sig, cont);
    if (cont==N)
    {        for (i=0; i<2; i++)
    {            pf=wait(&status);
            if ((char)status==0)
                printf("term. %d con stato%d\n", pf,
status>>8);
            else
                printf("term. %d inv. (segnale %d)\n",
pf, (char)status);            }
            exit(0);
    }    return;
}
```

```
void timeout(int sig)
{ printf("figlio%d: scaduto
timeout!\n");
  kill(PID1, SIGKILL);
  exit(0);
}
```

Esercizio 5: system Calls

Si realizzi un programma, che, utilizzando le system call del sistema operativo UNIX, soddisfi le seguenti specifiche:

Sintassi di invocazione:

Esame C N

Significato degli argomenti:

- Esame è il nome del file eseguibile associato al programma.
- N e` un intero non negativo.
- C e` una stringa che rappresenta il nome di un file eseguibile (per semplicita`, si supponga che il direttorio di appartenenza del file **C** sia nel PATH).

Specifiche:

Il processo iniziale (PO) deve creare 1 processo figlio P1 che, a sua volta crea un proprio figlio P2 . Si deve quindi ottenere una gerarchia di 3 processi: PO (padre), P1 (figlio) e P2 (nipote).

- **Il processo P2**, una volta creato, passa ad eseguire il comando C.
- **Il processo P1** , dopo aver generato P2 , deve mettersi in attesa di uno dei 2 eventi seguenti:
 1. la ricezione di un segnale dal padre, oppure
 2. la terminazione di P2.

Nel primo caso (ricezione di un segnale da PO) P1 termina forzatamente l'esecuzione di P2 e poi termina.

Nel secondo caso (terminazione di P2), P1 invia un segnale al padre PO e successivamente termina trasferendo a PO il pid di P2

- **Il processo PO**, dopo aver generato P1, entra in un ciclo nel quale, ad ogni secondo, incrementa un contatore K; se K raggiunge il valore N, PO invia un segnale al figlio P1 e termina. Nel caso in cui PO riceva un segnale da P1 durante l'esecuzione del ciclo, prima di terminare dovrà stampare lo stato di terminazione del figlio e successivamente terminare.

Soluzione dell'esercizio

```
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <stdlib.h>
#define dim 10

int P1, P2;
int status;
void GP(int sig); //gestore padre
void GF(int sig); //gestore segnali figlio P1

main(int argc , char *argv[])
{
    int N, K=0;
    char C[dim];
    if( argc != 3 ) {
        printf( "sintassi: %s      <nome_comando>   <N>\n",
            argv[0] );
        exit(2);
    }
    strcpy(C,argv[1]);
    N=atoi(argv[2]);
```

```
signal(SIGUSR1, GP) ;
P1=fork();
if (P1==0)//figlio
{
    printf("sono il figlio ..\n");
    signal(SIGUSR1, GF);
    signal(SIGCHLD, GF);
P2=fork();
    if (P2==0) //nipote
{
    execlp(C, C, NULL);
    perror("attenzione: exec fallita!");
    exit(1);
}
pause();      //P1: attesa evento
exit(0);
}
else          //padre
{
for (K=0; K<N; K++)
    sleep(1);
printf("padre P0: esaurito ciclo di conteggio - segnale a
P1 ..\n");
    kill(P1, SIGUSR1);
}
exit(0); /* fine padre */
```

```
void GP(int sig)
{ int pf, status;
  printf( "P0 (PID: %d) :RICEVUTO SIGUSR1\n", getpid());
  pf=wait(&status);
  if ((char)status==0)
    printf("PADRE: valore trasferito da P1 (pid
di P2): %d\n", status>>8);
  else
    printf("PADRE: la terminazione di %d involontaria
(per
    segnale %d)\n", pf, (char)status);
  exit(1);
}
```

```
void GF(int sig)
{ int pf, status;
  if (sig==SIGCHLD) //P2 terminato
  { printf( "P1 (PID: %d):RICEVUTO SIGCHLD-> esecuzione
di P2

              terminata\n", getpid());
    pf=wait(&status);
    if ((char)status==0)

                  printf("P1: terminazione di %d con
stato %d\n", pf, status>>8);
    else          printf("P1: terminazione di %d
involontaria (segnale %d)\n", pf, (char)status);
    kill(getppid(), SIGUSR1);
    exit(pf); //trasferimento pid di P2 al padre P0
  }
  else //segnale da P0
  { printf( "P1 (PID: %d):RICEVUTO SIGUSR1-> uccido
P2\n", getpid());
    kill(P2, SIGKILL);
    exit(0);
  }
}
```

monitor

Esercizio 6: monitor

Si consideri una lavanderia “a gettone”, nella quale sono disponibili 2 tipi di macchine: **lavatrici** e **asciugatrici**.

Il numero di lavatrici installate nella lavanderia è L , ognuna caratterizzata da una **potenza assorbita** di **PL** KW (kilowatt).

Il numero di asciugatrici installate nella lavanderia è A , ognuna caratterizzata da una **potenza assorbita** di **PA** KW.

Il contratto di erogazione di energia elettrica della lavanderia prevede un massimo di **P_MAX** KW di potenza impegnata; questo implica che in ogni istante la somma delle potenze dei vari dispositivi attivati non debba superare il valore P_MAX.

Ogni cliente che entra nella lavanderia porta con sè un carico di biancheria da lavare e asciugare, e ha quindi il seguente comportamento:

1. **Lavaggio**: utilizzo di una lavatrice per il lavaggio della biancheria; in particolare questa fase si articola in 2 passi:
 1. **Acquisizione** della lavatrice, inserimento del carico e attivazione;
 2. **Estrazione** del carico lavato e rilascio della lavatrice.
2. **Asciugatura**: uso di un'asciugatrice per asciugare la biancheria precedentemente lavata; anche questa fase si articola in 2 passi:
 - a. **Acquisizione** dell'asciugatrice, inserimento del carico e attivazione;
 - b. **Estrazione** del carico asciugato e rilascio dell'asciugatrice.

Si assuma, per semplicità, che il carico di ogni cliente possa essere sempre contenuto in una sola lavatrice/asciugatrice.

I clienti possono essere di due tipi:

- **Abbonati**: clienti che hanno sottoscritto un contratto di abbonamento con la lavanderia;
- **Non abbonati**.

Si realizzi un programma java nel quale i **clienti** siano rappresentati da thread concorrenti e che, utilizzando il **monitor** e le **variabili condizione**, regoli l'uso delle macchine della lavanderia tenendo conto dei vincoli dati e, inoltre, dei seguenti vincoli di **priorità**:

- Nell'accesso agli elettrodomestici, venga data la **priorità agli utenti delle asciugatrici**;
- Nell'acquisizione di un particolare tipo di elettrodomestico (lavatrice o asciugatrice) vengano **privilegiati i clienti abbonati**.

Impostazione

1. Quali processi?
2. Qual è la struttura di ogni processo?
3. Definizione del monitor per gestire la risorsa: **lavanderia**
4. Definizione delle procedure "entry"
5. Definizione del programma concorrente

Impostazione

1. Quali processi? Gli utenti della lavanderia

- abbonati
- non abbonati

2. Quale struttura per i processi ?

```
lavanderia.inizioLavaggio(this);  
    <uso lavatrice>  
  
lavanderia.fineLavaggio(this);  
  
..  
  
lavanderia.inizioAsciugatura(this);  
    <uso asciugatrice>  
lavanderia.fineAsciugatura(this);
```

2. Definizione threads

```
public class Cliente extends Thread
{ private int id, tipo; //tipo 1=ABBONATO, 0=NON ABBONATO
private Lavanderia lavanderia;

public Cliente(int i,int t,Lavanderia lav)
{   id=i;
    tipo=t;
    lavanderia=lav;
}

public int getTipo()
{ return tipo;
}

public int getID()
{   return id;
}
```

```
public void run()
{
    lavanderia.inizioLavaggio(this);
    sleep((int)(4000*Math.random()));
    lavanderia.fineLavaggio(this);
    sleep((int)(1000*Math.random()));
    lavanderia.inizioAsciugatura(this);
    sleep((int)(5000*Math.random()));
    lavanderia.fineAsciugatura(this);

}
} // fine Cliente
```

3. Monitor

Sincronizzazione :

- Abbonati e non sono soggetti a vincoli di sincronizzazione diversi
 - possibilità di attesa per acquisizione lavatrice e asciugatrice per abbonati e non.
- prevedo 4 condition:

```
private Condition[] lavaggio=new Condition[2]; //code di clienti (standard,abbonati) in attesa del lavaggio  
private Condition[] asciugatura=new Condition[2]; //code di clienti (standard,abbonati) in attesa dell'asciugatura
```

3. Definizione del monitor

```
public class Lavanderia
{
    private final int L,PL,A,PA,PMAX;
    private final int ABBONATI=1;
    private final int STANDARD=0;
    private Lock lock=new ReentrantLock();
    private Condition[] lavaggio=new Condition[2];
    private Condition[] asciugatura=new Condition[2];
    private int[] slavaggio=new int[2];
    private int[] sasciugatura=new int[2];
    //numero di lavatrici e asciugatrici occupate e consumo
    attuale in KW:
    private int occL=0,occA=0,consumo=0;
```

```
public Lavanderia(int l,int pl,int a,int pa,int pmax)
{
    L=l;
    PL=pl;
    A=a;
    PA=pa;
    PMAX=pmax;
    for(int i=0;i<2;i++)
    {
        lavaggio[i]=lock.newCondition();
        asciugatura[i]=lock.newCondition();
        slavaggio[i]=0;
        sasciugatura[i]=0;
    }
}
```

```
public void inizioLavaggio(Cliente c) throws  
InterruptedException  
{ lock.lock();  
int tipoCliente=c.getTipo();  
try{  
    if(tipoCliente==ABBONATI) // lavaggio abbonato  
    { while(occL==L || (consumo+PL)>PMAX ||  
          sasciugatura[STANDARD]>0 ||  
          sasciugatura[ABBONATI]>0)  
        { slavaggio[ABBONATI]++;  
          lavaggio[ABBONATI].await();  
          slavaggio[ABBONATI]--;  
        }  
    occL++;  
    consumo+=PL;  
}  
// continua..
```

```
else // lavaggio non abbonato
{
    while(occL==L || (consumo+PL)>PMAX ||
          sasciugatura[STANDARD]>0 || sasciugatura[ABBONATI]>0
          || slavaggio[ABBONATI]>0)
    {
        slavaggio[STANDARD]++;
        lavaggio[STANDARD].await();
        slavaggio[STANDARD]--;
    }
    occL++;
    consumo+=PL;
}
}finally {lock.unlock();    }
}
```

```
public void fineLavaggio(Cliente c) throws  
InterruptedException  
{ lock.lock();  
    occL--;  
    consumo-=PL;  
    if (sasciugatura[ABBONATI]>0)  
        asciugatura[ABBONATI].signalAll();  
    if (sasciugatura[STANDARD]>0)  
        asciugatura[STANDARD].signalAll();  
    if (sasciugatura[ABBONATI]==0 &&  
        sasciugatura[STANDARD]==0)  
        if(slavaggio[ABBONATI]>0)  
            lavaggio[ABBONATI].signal();  
        else if (slavaggio[STANDARD]>0)  
            lavaggio[STANDARD].signal();  
    lock.unlock();  
}
```

```
public void inizioAsciugatura(Cliente c) throws  
InterruptedException  
{ lock.lock();  
int tipoCliente=c.getTipo();  
try {  
    if(tipoCliente==ABBONATI) // asciugatura abbonati  
    { while(occA==A || (consumo+PA)>PMAX)  
        { sasciugatura[ABBONATI]++;  
        asciugatura[ABBONATI].await();  
        sasciugatura[ABBONATI]--;  
        }  
    occA++;  
    consumo+=PA;  
    }  
}
```

```
else // asciugatura non abbonati
{ while(occA==A || (consumo+PA)>PMAX ||  

       sasciugatura[ABBONATI]>0)
{   sasciugatura[STANDARD]++;
    asciugatura[STANDARD].await();
    sasciugatura[STANDARD]--;
}
occA++;
consumo+=PA;
}
} finally { lock.unlock(); }
}
```

```
public void fineAsciugatura(Cliente c) throws  
InterruptedException  
{ lock.lock();  
occA--;  
consumo-=PA;  
if (sasciugatura[ABBONATI]>0)  
    asciugatura[ABBONATI].signal();  
else if (sasciugatura[STANDARD]>0)  
    asciugatura[STANDARD].signal();  
if (sasciugatura[ABBONATI]==0 &&  
    sasciugatura[STANDARD]==0)  
{ if(slavaggio[ABBONATI]>0)  
    lavaggio[ABBONATI].signalAll();  
if (slavaggio[STANDARD]>0)  
    lavaggio[STANDARD].signalAll();  
}  
lock.unlock();  
}  
}// fine Lavanderia
```

Definizione main

```
public class lavatrici
{ public static void main(String[] args)
{   int POTENZA_MAX=90;
    int PL=10; int PA=20;
    int L=5; int A=4;
    int nclientiST=10;  int nclientiABB=15;
    Lavanderia lav=new Lavanderia(L,PL,A,PA,POTENZA_MAX);
    Cliente[] clientiST=new Cliente[nclientiST];
    Cliente[] clientiABB=new Cliente[nclientiABB];
    for(int i=0;i<nclientiST;i++)
        clientiST[i]=new Cliente(i,0,lav);
    for(int i=0;i<nclientiABB;i++)
        clientiABB[i]=new Cliente(i,1,lav);
    for(int i=0;i<nclientiST;i++)
        clientiST[i].start();
    for(int i=0;i<nclientiABB;i++)
        clientiABB[i].start();
}
}
```

File comandi bash

Esercizio 7: file comandi

Si scriva un file comandi in shell di Linux che abbia l'interfaccia:

findNewerFiles <targetDir> <report> <date>

dove

- **<targetDir>** è il nome assoluto di un directory esistente nel filesystem,
- **<report>** il nome assoluto di un file di testo non esistente nel filesystem
- **<date>** una data espressa nel formato “yyyy-mm-dd”.

Si svolgano gli opportuni controlli sugli argomenti di invocazione del file comandi.

Il compito del file comandi è quello di **esplorare completamente la gerarchia individuata dal direttorio <targetDir>**.

Per ogni directory esplorato, il programma deve **cercare tutti i file ordinari** (non directory, non dispositivo e non link) **la cui data di modifica sia più recente di <date>**.

In tal caso, il programma deve **scrivere sul file <report>** il nome assoluto del file secondo la seguente logica:

- se il file considerato è stato modificato in un anno più recente rispetto a quello riportato in <date>, su <report> andrà scritta la stringa "anno<anno_modifica> -<nomeAssolutoFile>"
- se il file considerato è stato modificato nel medesimo anno ma in un mese più recente rispetto a quello riportato in <date>, su <report> andrà scritta la stringa "mese<mese_modifica> -<nomeAssolutoFile>"
- se il file considerato è stato modificato nel medesimo anno e nel medesimo mese, ma in un giorno più recente rispetto a quello riportato in <date>, su <report> andrà scritta la stringa "giorno<giorno_modifica> - <nomeAssolutoFile>"

Si suggerisce l'utilizzo dei **comandi bash predefiniti**:

- **stat**, con opportuno parametro (--format=%z), per reperire la data di modifica di un file nel formato voluto;

```
$ stat --format=%z pippo  
2010-11-15 04:02:38.000000000 -0800
```

- **cut**, con opportuni parametri, per l'estrazione di parti da una stringa, come nel caso delle elaborazioni necessarie su <date>.

Esempio di soluzione

- 2 file:
 - **findNewerFiles.sh:** controllo argomenti, settaggio path e invocazione del file ricorsivo:
 - **findNewerFiles_rec.sh:**
 - Esecuzione ricorsiva a partire dalla radice della gerarchia

findNewerFiles.sh

```
#!/bin/bash

if test $# -ne 3
then
    echo "usage:$0 <scrDir> <reportFile> <yyyy-mm-dd>"
    exit 1
fi

case $1 in
/*) ;;
*) echo "$1 is not an absolute directory"
    exit 4;;
esac

if ! test -d "$1"
then
echo "$1 is not a valid directory"
exit 5
fi
```

```
case $2 in
/*) ;;
*) echo "$2 is not an absolute file"
    exit 4;;
esac

case $3 in
????-??-??) ;;
*) echo "Date $3 should have the format \"yyyy-mm-dd\""
    exit 4;;
esac

anno=`echo $3 | cut -d ' ' -f1| cut -d '-' -f1`
mese=`echo $3 | cut -d ' ' -f1 | cut -d '-' -f2`
giorno=`echo $3 | cut -d ' ' -f1 | cut -d '-' -f3`

oldpath=$PATH
PATH=$PATH:`pwd`  

    findNewerFiles_rec.sh $1 $2 $anno $mese $giorno
PATH=$oldpath
```

findNewerFiles_rec.sh

```
#!/bin/sh
#$1: directory nella quale andare in ricorsione
#$2: nome assoluto file di report
#$3: anno
#$4: mese
#$5: giorno
cd "$1"
for f in *
do
    if test -d "$f"
        then
            $0 $1/"$f" $2 $3 $4 $5
    elif test -f "$f"
        then
            anno=`stat --format=%z "$f" | cut -d ' ' -f1 | cut -d '-' -f1`
            mese=`stat --format=%z "$f" | cut -d ' ' -f1 | cut -d '-' -f2`
            giorno=`stat --format=%z "$f" | cut -d ' ' -f1 | cut -d '-' -f3`
```

```
if test $anno -gt $3; then
    echo anno_$anno - `pwd`/"$f">>> $2
elif test $mese -gt $4 -a $anno -eq $3; then
    echo mese_$mese - `pwd`/"$f">>> $2
elif test $mese -eq $4 -a $anno -eq $3 -a
$giorno -gt -$5; then
    echo giorno_$giorno - `pwd`/"$f">>> $2
fi
fi
done
```

Esercizio 8: file comandi

Si realizzi un file comandi Unix con la seguente interfaccia:

copy.sh <dir> <string> <dest>

- <dir> e <dest> direttori assoluti esistenti nel filesystem;
- <string> una stringa.

Dopo aver effettuato tutti gli opportuni controlli sui parametri in ingresso, il file comandi deve cercare, in ciascun sottodirettorio di dir, tutti i **file ordinari nelle cui prime 10 righe compaia <string> almeno una volta**. Per ciascun file così trovato all'interno di un sottodirettorio, **si copi tale file** in un opportuno sottodirettorio di <dest> di nome **<dest>/N**, cioè un sottodirettorio di <dest> il cui nome sia uguale al **numero effettivo** di occorrenze di <string> trovate nelle prime 10 righe del file.

Ad esempio, supponendo di invocare il comando con
copy.sh /home/user pdf /home/backup

e di avere la seguente condizione su filesystem:

/home/user/prova.txt (3 occ. di pdf nelle prime 10 righe)
/home/user/proval.txt
/home/user/pippo.xml (7 occ. di pdf nelle prime 10 righe)
/home/user/dir1/prova.txt
/home/user/dir1/prova.pdf

il file comandi creerà e riempirà il direttorio di backup in questo modo:

/home/backup/7/pippo.xml
/home/backup/3/prova.txt

Esempio di Soluzione

- 2 file:
 - **copy.sh:** controllo argomenti, settaggio path e invocazione del file ricorsivo:
 - **copy_rec.sh:**
 - Esecuzione ricorsiva a partire dalla radice della gerarchia

Copy.sh

```
#!/bin/sh

# Controllo parametri
if test $# -ne 3
then
    echo "usage:$0 <dir> <string> <dest>"
    exit 1
fi
case $1 in
    /*) ;;
    *) echo "$1 is not an absolute directory"
       exit 4;;
esac
if ! test -d "$1"
then
    echo "$1 is not a valid directory"
    exit 5
fi
```

```
#..continua
case $3 in
    /*) ;;
    *) echo "$3 is not an absolute directory"
        exit 4;;
esac
if ! test -d "$3"
then
    echo "$3 is not a valid directory"
    exit 5
fi
```

```
# Invocazione script ricorsivo:
oldpath=$PATH
PATH=$PATH:`pwd`\
copy_rec.sh "$1" $2 "$3"
PATH=$oldpath
```

Copy_rec.sh

```
#!/bin/sh
cd "$1"
for f in *
do
    if test -d "$f"
    then
        $0 "$f" $2 "$3"
    elif test -f "$f"
    then
        count=`head -n 10 "$f" | grep -o "$2" | wc -l`
        if test $count -gt 0
        then
            if ! test -d "$3"/$count
            then
                mkdir "$3"/$count
            fi
            cp "$f" "$3"/$count
        fi
    fi
done
```

Ulteriori Esempi shell

Esempio 1

- Creare uno script che abbia la sintassi:
`./ps_monitor.sh [N]`
- Lo script:
 - in caso di assenza dell'argomento, deve mostrare i processi di tutti gli utenti (compresi quelli senza terminale di controllo) con anche le informazioni sul nome utente e ora di inizio;
 - Se viene passato come argomento un intero (N) deve mostrare i primi N processi

NOTA: non tutte le righe prodotte in output da ps hanno contenuto informativo rilevante

Soluzione

```
#!/bin/bash
if [ -z $1 ] # restituisce 1 se il primo param.
            # e' una stringa vuota
then
    ps aux
else
    ps aux | head -n `expr $1 + 1`
#consideriamo che c'e' anche una riga di intest.
fi
```

Esempio 2

- Creare uno script che abbia la sintassi

```
./lines_counter.sh <directory> [up|down]
```

- Lo script deve elencare i file contenuti nella directory con relativo numero di linee, ordinati in senso crescente (up) o decrescente(down)

NOTA: controllare:

- Che il primo argomento sia effettivamente una directory
- Che il secondo argomento sia la stringa up o down

Soluzione

```
#!/bin/bash
if [ $# -ne 2 ] #sintassi sbagliata
then
    echo "SINTASSI: lines_counter.sh <directory> [up|down]"
    exit 1 #uscita anomala
fi
if [ -d $1 ] #vero se $1 è una directory
then
    if [ $2 = "up" ]
    then
        wc -l $1/* | sort -n
#1. viene espansa la lista di tutti i file presenti in $1
#2. su ogni elemento viene eseguito il conteggio
#3. viene effettuato l'ordinamento sui conteggi
```

```
elif [ $2 = "down" ]
then
    wc -l $1/* | sort -nr #come sopra, ma
                           l'ordinamento è inverso
else
    echo "ERROR: 'up' or 'down'"
    exit 2 #uscita anomala
fi
else
    echo "$1 should be an existent directory"
    exit 2 #uscita anomala
fi
```

Esempio 3

- Creare uno script che abbia la sintassi

```
./backup.sh <nomefile> <nomebackup>
```

- Se il file è una directory, lo script deve:
 - creare una sottodirectory(rispetto a livello corrente) di nome: <nomefile>_<nomebackup>
 - copiare ricorsivamente in essa il contenuto della directory
- Se il file è un file normale, lo script deve crearne 5 copie di nome <nomefile>*i<nomebackup> i=1..5

Soluzione

```
#!/bin/bash

#IPOTESI: considero solo file e direttori nel dir.corrente

if [ $# -ne 2 ]
then
    echo "USAGE: backup.sh <filename> <backupstring>"
    exit 1
fi

if [ -d $1 ]
#-restituisce 1 se il primo parametro e' una directory
then
    cp -R $1 "$1_$2"
```

```
elif [ -f $1 ] #controlla che $1 sia un file normale
then
    for i in 1 2 3 4 5 #i cicla sugli el. della lista
    do
        cp $1 "$1*${i}*$2"
#i doppi apici proteggono l'espansione di * ma non di $
    done
else
    echo "$1 should be a valid directory or file"
fi
```